Clustered Serialization with Fuel

Martín Dias, Mariano Martinez Peck, Stéphane Ducasse, Gabriela Beatriz Arévalo

Clustered Serialization with Fuel

Martín Dias[1 3] Mariano Martinez Peck[1 2] Stéphane Ducasse[1] Gabriela Arévalo[4 5]

[1]RMoD Project-Team, Inria Lille–Nord Europe / Université de Lille 1

[2]Ecole des Mines de Douai, [3]Universidad de Buenos Aires, [4]Universidad Abierta Interamericana [5]CONICET

{tinchodias, marianopeck, gabriela.b.arevalo}@gmail.com, stephane.ducasse@inria.fr

Abstract

Serializing object graphs is an important activity since objects should be stored and reloaded on different environments. There is a plethora of frameworks to serialize objects based on recursive parsing of the object graphs. However such approaches are often too slow. Most approaches are limited in their provided features. For example, several serializers do not support class shape changes, global references, transient references or hooks to execute something before or after being stored or loaded. Moreover, to be faster, some serializers are not written taking into account the object-oriented paradigm and they are sometimes even implemented in the Virtual Machine hampering code portability. VM-based serializers such as ImageSegment are difficult to understand, maintain, and fix. For the final user, it means a serializer which is difficult to customize, adapt or extend to his own needs.

In this paper we present a general purpose object graph serializer based on a pickling format and algorithm. We implement and validate this approach in the Pharo Smalltalk environment. We demonstrate that we can build a really fast serializer without specific VM support, with a clean object-oriented design, and providing most possible required features for a serializer. We show that our approach is faster that traditional serializers and compare favorably with ImageSegment as soon as serialized objects are not in isolation.

Keywords Object-Oriented Programming and Design » Serializer » Object Graphs » Pickle Format » Smalltalk

1. Introduction

In the Object-Oriented Programming paradigm, since objects point to other objects, the runtime memory is an object graph. This graph of objects lives while the system is running and dies when the system is shutdown. However, sometimes it is necessary, for example, to backup a graph of objects into a non volatile memory to load it back when necessary, or to export it so that the objects can be loaded in a different system. The same happens when doing migrations or when communicating with different systems.

Approaches and tools to export object graphs are needed. An approach must scale to large object graphs as well as be efficient. However, most of the existing solutions do not solve this last issue properly. This is usually because there is a trade-off between speed and other quality attributes such as readability/independence from the encoding. For example, exporting to XML [17] or JSON [9] is more readable than exporting to a binary format, since one can open it and edit it with any text editor. But a good binary format is faster than a text based serializer when reading and writing. Some serializers like *pickle* [14] in Python or *Google Protocol Buffers* [15] that let one choose between text and binary representation. Five main points shape the space of object serializers

1. Serializer *speed* is an important aspect since it enables more extreme scenarios such as saving objects to disk and loading them only on demand at the exact moment of their execution [3, 10].
2. Serializer *portability and customization*. Since many approaches are often too slow, Breg and Polychronopoulos advocate that object serialization should be done at the virtual machine level [4]. However, this implies non portability of the virtual machine and difficult maintenance. In addition, moving behavior to the VM level usually means that the serializer is not easy to customize or extend.
3. Another usual problem is *class* changes. For example, the class of a saved object can be changed after the object is saved. At writing time, the serializer should store all the necessary information related to class shape to deal with these changes. At load time, objects must be updated in case it is required. Many object serializers are limited regarding this aspect. For example, the Java Serializer [8] supports adding or removing a method or a field, but does not support the modification of an object's hierarchy nor the removing of the implementation of the Serializable interface.
4. *Loading policies*. Ungar [18] claims that the most important and complicated problem is not to detect the sub-

graph to export, but to detect the implicit information of the subgraph that is necessary to correctly load back the exported subgraph in another system. Examples of such information are (1) whether to export an actual value or a counterfactual initial value, or (2) whether to create a new object in the new system or to refer to an existing one. In addition, it may be necessary that certain objects run some specific code once they are loaded in a new system.

5. *Uniformity of the solution.* Serializers are often limited to certain kind of objects they save. For example MsgPack only serializes booleans, integers, floats, strings, arrays and dictionaries. Now in dynamic programming languages *e.g.,* Smalltalk, methods and classes as first class objects – user's code is represented as objects. Similarly the execution stack and closures are objects. The natural question is if we can use serializers as code management system underlying mechanism. VisualWorks Smalltalk introduced a pickle format to save and load code called Parcels [12]. However, such infrastructure is more suitable for managing code than a general purpose object graph serializer.

This paper presents Fuel, an open-source general purpose framework to serialize and deserialize object graphs using a pickle format which clusters similar objects. We show in detailed benchmarks that we have the best performance in most of the scenarios we are interested in. For example, with a large binary tree as sample, Fuel is 16 times faster loading than its competitor SmartRefStream, and 7 times faster writing. We have implemented and validated this approach in the Pharo Smalltalk Environment [2].

The pickle format presented in this paper is similar to the one of Parcels [12]. However, Fuel is not focused in code loading and is highly customizable to cope with different objects. In addition, this article demonstrates the speed improvements gained in comparison to traditional approaches. We demonstrate that we can build a fast serializer without specific VM support, with a clean object-oriented design, and providing most possible required features for a serializer.

The main contributions of the paper are:

1. Description of our pickle format and algorithm.
2. Description of the key implementation points.
3. Evaluation of the speed characteristics.
4. Comparison of speed improvements with other serializers.

The remainder of the paper is structured as follows: Section 2 provides a small glossary for the terms we use in the paper. Section 3 enumerates common uses of serialization. Features that serializers should support are stressed in Section 4. In Section 5 we present our solution and an example of a simple serialization which illustrates the pickling format. Fuel key characteristics are explained in details in Section 6. A large amount of benchmarks are provided in Section 8. Finally, we discuss related work in Section 9 and we conclude in Section 10.

2. Glossary

To avoid confusion, we define a *glossary* of terms used in this paper.

Serializing. It is the process of converting the whole object graph into a sequence of bytes. We consider the words *pickling* and *marshalling* as synonyms.

Materializing. It is the inverse process of serializing, *i.e.,* regenerate the object graph from a sequence of byes. We consider the words *deserialize*, *unmarshalling* and *unpickling* as synonyms.

Object Graph Serialization. We understand the same for *object serialization*, *object graph serialization* and *object subgraph serialization*. An object can be seen as a subgraph because of its pointers to other objects. At the same time, everything is a subgraph if we consider the whole memory as a large graph.

Serializer. We talk about serializer as a tool performing both operations: serializing and materializing.

3. Serializer Possible Uses

We will mention some possible uses for an object serializer.

Persistency and object swapping. Object serializers can be used *directly* as a light persistency mechanism. The user can serialize a subgraph and write it to disk or secondary memory [11]. After, when needed, it can be materialized back into primary memory [3, 10]. This approach does not cover all the functionalities that a database provides but it can be a good enough solution for small and medium size applications. Besides this, databases normally need to serialize objects to write them to disk [5].

Remote Objects. In any case of remote objects, *e.g.,* remote method invocation and distributed systems [1, 6, 19], objects need to be passed around the network. Therefore, objects need to be serialized before sending them by the network and materialized when they arrive to destination.

Web Frameworks. Today's web applications need to store state in the HTTP sessions and move information between the client and the server. To achieve that web frameworks usually use a serializer.

Version Control System. Some dynamic programming languages *e.g.,* Smalltalk consider classes and methods as first-class objects. Hence, a version control system for Smalltalk deals directly with objects, not files as in other programming languages. In this case, such tool needs to serialize and materialize objects. This is the main purpose behind Parcels [12].

Loading methods without compilation. Since classes and methods are objects, a good serializer should be enable the loading of code without the presence of a compiler. This enable binary deployment, minimal kernel, and faster loading time.

4. Serializer Features

In this section we enumerate elements to analyze a serializer. We start with more abstract concerns and then follow with more concrete challenges. We use these criteria to discuss and evaluate our solution in Section 7.

4.1 Serializer concerns

Below we list general aspects to analyze on a serializer.

Performance. In almost every software component, time and space efficiency is a wish or sometimes even a requirement. It does become a need when the serialization or materialization is frequent or when working with large graphs. We can measure both speed and memory usage, either serializing and materializing, as well as the size of the result stream. We should also take into account the initialization time, which is important when doing frequent small serializations.

Completeness. It refers to what kind of objects can the serializer handle. It is clear that it does not make sense to transport instances of some classes, like FileStream or Socket. Nevertheless, serializers often have limitations that restrict use cases. For example, an apparently simple object like a SortedCollection usually represents a problematic graph to store: it references a block closure which refers to a method context, and most serializers does not support transporting them, often due to portability reasons.

Besides, in comparison with other popular environments, the object graph that one can serialize in Smalltalk is much more complex. This is because it reifies elements like the metalevel model, methods, block closures, and even the execution stack. Normally there are two approaches: for *code* sharing, and for *plain objects* sharing.

Portability. There are two aspects related to portability. One is related to the ability to use the same serializer in different dialects of the same language. For example, in Smalltalk one would like to use the same, or at least almost the same, code for different dialects. The second aspect is related to the ability of be able to materialize in a dialect or language a stream which was serialized in another language. This aspect brings even more problems and challenges to the first one.

As every language and environment has its own particularities, there is a trade-off between portability and completeness. Float and BlockClosure instances often have incompatibility problems.

For example, Action Message Format (AMF), Google Protocol Buffers, Oracle Coherence*Web, Hessian, are quite generic and there are implementations in several languages. In contrast, SmartRefStream in Squeak and Pharo, and pickle [14] in Python are not designed with this goal in mind. They just work in the language they were defined.

Abstraction capacity. Although it is always possible to store an object, concretely serializing its variables, such a low-level representation, is not acceptable in some cases. For example, an OrderedCollection can be stored either as its internal structure or just as its sequence of elements. The former is better for an accurate reconstruction of the graph. The second is much more robust in the sense of changes to the implementation of OrderedCollection. Both alternatives are valid, depending on the use case.

Security. Materializing from an untrusted stream is a possible security problem in the image. When loading a graph, some kind of dangerous object can enter to the environment. The user may want to control in some way what is being materialized.

Atomicity. We have this concern expressed in two parts: for saving and for loading. As we know, the image is full of mutable objects *i.e.,* that changes their state over the time. So, while serialization process is running. It is desired that when serializing such mutable graph it is written an atomic snapshot of it, and not a potential inconsistent one. On the other hand, when loading from a broken stream and so it can not complete its process. In such case, no secondary effects should affect the environment. For example, there can be an error in the middle of the materialization which means that certain objects have been already materialized.

Versatility. Let us assume a class is referenced from the graph to serialize. Sometimes we may be interested in storing just the name of the class because we know it will be present when materializing the graph. However, sometimes we want to really store the class with full detail, including its method dictionary, methods, class variables, etc. When serializing a package, we are interested in a mixture of both: for external classes, just the name, for the internal ones, full detail.

This means that given an object graph living in the image, there is not an unique way of serializing it. A serializer may offer dynamic or static mechanisms to the user to customize this behavior.

4.2 Serializer challenges

The following is a list of concrete issues and features we consider in serializers:

Maintaining identity. When serializing an object we actually serialize an object graph. However, we usually do not want to store the whole transitive closure of references of the object. We know (or we hope) that some objects will be present when loading, so we want just to store *external references*, *i.e.,* store the necessary information to be able to look those objects in the environment while materializing.

In Figure 1 we have an example of a method, *i.e.,* a CompiledMethod instance and its subgraph of references. Suppose that the idea is to store just the method. We know that the class and the global binding will be present on loading. Therefore, we just reference them by encoding their names.

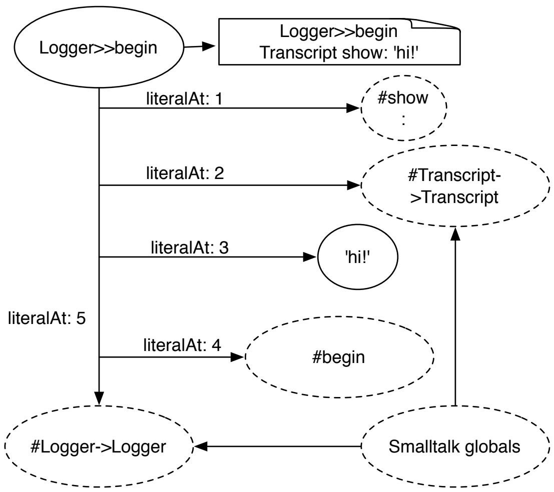

Figure 1. Serializing a method while maintaining identity of its referenced classes and globals.

In other words, in this example we want to serialize a method, maintaining identity of its class, the global Transcript, and the symbols. Consequently, when materializing, the only instances that should be created are the compiled method and the string. The rest will be looked up in the environment.

Transient values. Sometimes objects have temporal state that we do not want to store, and we want an initial value when loading. A typical case is serializing an object that has an instance variable with a lazy-initialized value. Suppose we prefer not to store the actual value. In this sense, declaring a variable as *transient* is a way of delimiting the graph to serialize. We are breaking the iteration through that reference.

There are different levels of transient values:

Instance. When only one particular object is transient. All objects in the graph that are referencing to such object will be serialized with a nil in their instance variable that points to the transient object.

Class. Imagine we can define that a certain class is transient in which case all its instances are considered transient.

Instance variable names. This is the most common case. The user can define that certain instance variables of a class have to be transient. That means that all instances of such class will consider those instance variables as transient.

List of objects. The ability to consider an object to be transient only if it is found in a specific list of objects. The user should be able to add and remove elements from that list.

Cyclic object graphs and duplicates. Since commonly the object graph to serialize has cycles, it is important to detect them and take care to preserve objects identity. Supporting that means decreasing the performance and increasing the memory usage. This is because each iterated object in the graph should be temporally collected: it is necessary to check whether each object has been already processed or not.

Class shape change tolerance. Often we need to load instances of a class in an environment where its definition has changed. The expected behavior may be to adapt the old-shaped instances automatically when possible. We can see some examples of this in Figure 2. For instance variable position change, the adaptation is straightforward. For example, version v2 of Point changes the order between the instance variables x and y. For the variable addition, an easy solution is to fill with nil. Version v3 adds instance variable distanceToZero. If the serializer also lets one to write custom messages to be sent by the serializer once the materialization is finished, the user can benefit from this hook to initialize the new instance variables to something different that nil.

In contrast to the previous examples, for variable renaming, the user must specify what to do. This can be done via hook methods, or more dynamically, via materialization settings.

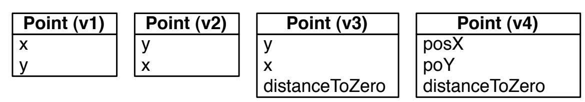

Figure 2. Several kinds of class shape changing.

There are even more kinds of changes such as adding, removing or renaming a method, a class or an instance variable, changing the superclass, etc. As far as we know, no serializer fully manage all these kinds of change. Actually, most of them have a limited number of supported change types. For example, the Java Serializer [8] supports the adding and the removing of a method or of a field, but does not support changing an object's hierarchy or removing the implementation of the Serializable interface.

Custom reading. When working with large graphs or when there is a large number of stored streams, it makes sense to read the serialized bytes in customized ways, not necessarily materializing all the objects as we usually do. For example, if there are methods written in the streams, we may want to look for references to certain message selectors. Maybe count how many instances of certain class we have stored. Maybe list the classes or packages defined or referenced from a stream. Or to extract any kind of statistics about the stored objects.

Partial loading. In some scenarios, especially when working with large graphs, it may be necessary to materialize only a part of the graph from the stream instead of the whole graph. Therefore, it is a good feature to simply get a subgraph with some holes filled with nil. In addition, the tool could support some kind of lazy loading.

Versioning. The user may need to load an object graph stored with a different version of the serializer. Usually this feature allows version checking so that future versions can detect that a stream was stored using another version, and act consequently: when possible migrate it, otherwise throw an error message.

Alternative output format. Textual or binary: serializers like *pickle* [14] in Python or Google Protocol Buffers [15] let the user choose between textual and binary representation. While developing, we can use the text based one, which is easy to see, inspect and modify. Then, at production time, we can switch to the faster binary format.

Alternative stream libraries. Usually, there are several packages of streams available for the same programming languages. For example, for Pharo Smalltalk there are Xtreams, FileSystem, and Nile. A design that supports alternative implementations is desired.

Graphical progress update. Object graphs can be huge and so, the user has to wait until the process end. Therefore, it is important to have the possibility to enable this feature and show graphically the processing of the graph.

5. Fuel

In this section we present Fuel, a new framework to serialize objects. Fuel is based on the hypothesis that fast loading is always preferable, even it implies a slower serialization process. Fuel clusters objects and separates relationships.

5.1 Pickle Formats

Pickle formats are efficient to support *transport*, *marshalling* or *serialization* of objects [16]. Before going any further we give a definition of pickle and give an example.

"*Pickling* is the process of creating a serialized representation of objects. Pickling defines the serialized form to include meta information that identifies the type of each object and the relationships between objects within a stream. Values and types are serialized with enough information to insure that the equivalent typed object and the objects to which it refers can be recreated. *Unpickling* is the complementary process of recreating objects from the serialized representation." (extracted from [16])

5.2 Pickling a rectangle

To present the pickling format and algorithm in an intuitive way, we show below an example of how Fuel stores a rectangle.

In Figure 3 we create a rectangle with two points that define the origin and the corner. A rectangle is created and then passed to the serializer as an argument. In this case the rectangle is the *root* of the graph which also includes the points that the rectangle references. The first step is to analyze the graph starting from the root. Objects are mapped to *clusters* following some criteria. In this example, the criteria is 'by class', but in other cases it is 'is global object' (it is at Smalltalk dictionary), or 'is an integer between 0 and 2^{16}'.

```
FLDemo >> serializeSampleRectangleOn: aFileStream

 | aRectangle anOrigin aCorner |
 anOrigin := 10@20.
 aCorner := 30@40.
 aRectangle := Rectangle origin: anOrigin corner: aCorner.

 (FLSerializer on: aFileStream) serialize: aRectangle.
```

Figure 3. Code snippet for our example to show how Fuel serialization works.

Finally, in Figure 4 we can see how the rectangle is stored in the stream. The graph is encoded in four main sections: header, vertexes, edges and trailer. The 'Vertexes' section collects the instances of the graph. The 'Edges' section contains indexes to recreate the references of the instances. The trailer encodes the root: a reference to the rectangle.

Even if the main goal of Fuel is to be fast in materialization, the benchmarks of Section 6 show that actually Fuel is fast for both serialization and materialization. In the next section, there are more details regarding the pickle format and how the clusters work.

6. Fuel Key Characteristics

In the previous section, we explained the pickle format behind Fuel, but that is not the only key aspect. The following is a list of important characteristics of Fuel:

Grouping objects in clusters. Tipically serializers do not group objects. Thus, each object has to encode its type at serialization and decode it at deserialization. This is not only an overhead in time but also in space. In addition, each object may need to fetch its class to recreate the instance.

The purpose of grouping similar objects is to reduce the overhead on the byte representation that is necessary to encode the *type* of the objects. The idea is that the type is encoded and decoded only once for all the objects of that type. Moreoever, if recreation is needed, the operations can be grouped.

The type of an object is sometimes directly mapped to its class, but the relation is not always one to one. For example, if the object being serialized is Transcript, the type that will be assigned is the one that represents global objects. For speed reason, we distinguish between positive SmallInteger and negative one. From a Fuel perspective they are from different types.

Clusters know how to encode and decode the objects they group. Clusters are represented in Fuel's code as classes. Each cluster is a class, and each of those classes has an associated unique identifier which is a number. Such ID is encoded in stream as we saw in Figure 4. It is written only once, at the begining of a cluster instance. At materialization time, the cluster ID is read and then the associated cluster is searched. The materializer then materializes all the objects it contains.

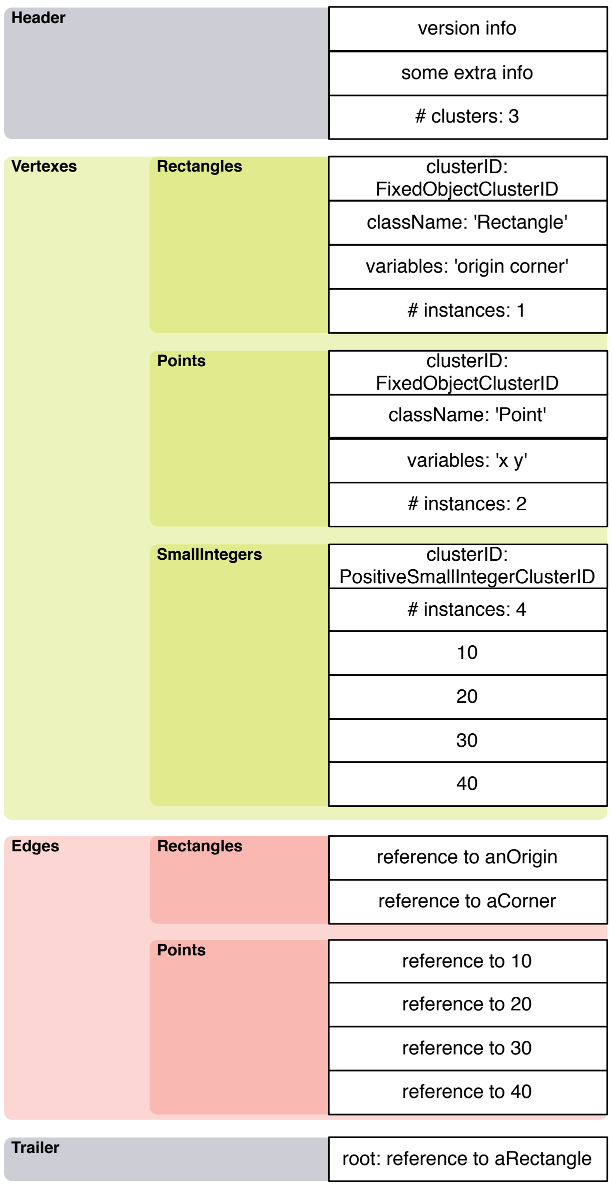

Figure 4. A graph example encoded with the pickle format.

Notice that what is associated to objects are cluster instances, not the cluster classes. The ID is unique per cluster. Some examples:

- PositiveSmallIntegerCluster is for positive instances of SmallInteger. Its ID is 4. A unique Singleton instance is used for all the objects grouped to this cluster.
- NegativeSmallIntegerCluster is for negative instances of SmallInteger. Its ID is 5. Again, it is singleton.
- FloatCluster is for Float instances. Its ID is 6. Again, it is singleton.
- FixedObjectCluster is the cluster for regular classes with indexable instance variables and that do not require any special serialization or materialization. Its ID is 10 and one instance is created for each class, *i.e.,* FixedObjectCluster has an instance variable referencing a class. One instance is created for Point and one for Rectangle.

If we analyze once again Figure 4, we can see that there is one instance of PositiveSmallIntegerCluster, and two instances of FixedObjectCluster, one for each class.

It is important to understand also that it is up to the cluster to decide what is encoded and decoded. For example, FixedObjectCluster writes into the stream a reference to the class whose instances it groups, and then it writes the instance variable names. In contrast, FloatCluster, PositiveSmallIntegerCluster or NegativeSmallIntegerCluster do not store such information because it is implicit in the cluster implementation.

In Figure 4 one can see that for small integers, the cluster directly writes the numbers 10, 20, 30 and 40 in the stream. However, the cluster for Rectangle and Point do not write the objects in the stream. This is because such objects are no more than just references to other objects. Hence, only their references are written in the 'Edges' part. In contrast, there are objects that contain self contained state, *i.e.,* objects that do not have references to other objects. Examples are Float, SmallInteger, String, ByteArray, LargePositiveInteger, etc. In those cases, the cluster associated to them have to write those values in the stream.

The way to specify custom serialization or materialization of objects is by creating specific clusters.

Analysis phase. The common approach to serialize a graph is to traverse it and while doing so to encode the objects into a stream. Fuel groups similar objects in clusters so it needs to traverse the graph and associate each object to its correct cluster. As explained, that fact significantly improves the materialization performance. Hence, Fuel does not have one single phase of traverse and writing, but instead two phases: analysis and writing.

The analysis phase has several responsibilities:

- It takes care of traversing the object graph and it associates each object to its cluster. Each cluster has a corresponding list of objects which are added there while they are analyzed.
- It checks whether an object has already been analyzed or not. Fuel supports cycles. In addition, this offers to write an object only once even if it is referenced from several objects in the graph.
- It gives support for global objects, *i.e.,* objects which are considered global and should not be written into the stream but instead put the minimal needed information to get it back the reference at materialization time. This is, for example, what is in Smalltalk globals. If there are objects in the graph referencing the instance Transcript we do not want to serialize it but instead just put a reference to it and at materialization get it back. In this case, just storing the global name is enough. The same happens with the Smalltalk class pools.

Once the analysis phase is over, the writing follows: it iterates over every cluster and for every cluster write its objects.

Stack over recursion. Most of the serializers use a depth-first traversal mechanism to serialize the object graph. Such mechanism consists of a simple recursion:

1. Take an object and look it up in a table.
2. If the object is in the table, it means that it has already been serialized. The, we take a reference from the table and write it down. If it is not present in the table, it means that the object has not been serialized and that its contents need to be written. After that, the object is serialized and a reference representation is written into the table.
3. While writing the contents, *e.g.,* instance variables of an object, the serializer can encounter simple objects such as instances of String, Float, SmallInteger, LargePositiveInteger, ByteArray or complex objects (objects which have instance variables that reference to other objects). In the latter case we start over from the first step.

This mechanism can consume too much memory depending on the graph, *e.g.,* its depth, the memory to hold all the call stack of the recursion can be too much. In addition, a Smalltalk dialect may limit the stack size.

In Fuel, we do not use the mentioned recursion but instead we use a stack. The difference is mainly in the last step of the algorithm. When an object has references to other objects, instead of following the recursion to analyze these objects, we just push such objects on a stack. Then we pop objects from the stack and analyze them. The routine is to pop and analyze elements until the stack is empty. In addition, to improve even more the speed, Fuel has its own SimpleStack class implementation.

That way, Fuel turns a recursive trace into depth-by-depth trace. With this approach the resulting stack size is much smaller and the memory footprint is smaller. At the same time, we increase serialization time by 10%.

Notice that Fuel makes it possible because of the separate analysis phase before the actual object serialization.

Two phases for writing instances and references. The encoding of objects is divided in two parts: (1) instances writing and (2) references writing. The first phase includes just the minimal information needed to recreate the instances *i.e.,* the vertexes of the graph. The second phase has the information to recreate references that connect the instances *i.e.,* the edges of the graph.

Iterative graph recreation. The pickling algorithm carefully sorts the instances paying attention to inter-depencencies.

During Fuel serialization, when a cluster is serialized, the amount of objects of such cluster is stored, as well as the total amount of objects of the whole graph. This means that at materialization time, Fuel know exactly the number of allocations (new objects) needed for each cluster. For example, one Fuel file contains 17 large integers, 5 floats, and 5 symbols, etc. In addition, for variable objects, Fuel also stores the size of such objects. So for example, it does not only know that there are 5 symbols but also that the first symbol is size 4, the second one 20, the third 6, etc.

Therefore, the materialize populates an object table with indices from 1 to N where N is the number of objects in the file. However, it does so in batch, spinning in a loop creating N instances of each class in turn, instead of determining which object to create as it walks a (flattened) input graph, as most of the serializers do.

Once that is done, the objects have been materialized, but updating the references is pending, *i.e.,* which slots refer to which objects. Again the materializer can spin filling in slots from the reference data instead of determining whether to instantiate an object or dereference an object id as it walks the input graph.

This is the main reason why materializing is much faster in Fuel than the other approaches.

Buffered write stream. A serializer usually receives two parameters from the user: the object graph to serialize and the stream where the former should be stored. Most of the time, such stream is a file stream or a socket stream. We know that writing to primary memory is much faster that writing to disc or to a network. Writing into a file stream or a network stream for every single object is terribly slow. Fuel uses an internal buffer to improve performance in that scenario. With a small buffer *e.g.,* 4096 elements, we get almost the same speed writing to a file or socket stream, as if we were writing in memory.

7. Fuel Features

In this section we analyze Fuel in accordance with the concerns and features defined in Section 4.

7.1 Fuel serializer concerns

Performance. We achieved an excellent time performance. This topic is extensively studied and explained in Section 8.

Completeness. We are close to say that Fuel deals with all kinds of objects. Notice the difference between being able to serialize and to get something meaningful while materializing. For example, Fuel can serialize and materialize instances of Socket, Process or FileStream, but it is not sure they will still be valid once they are materialized. For example, the operating system may have given the socket address to another process, the file associated to the file stream may have been removed, etc.

There is no magic. Fuel provides the infrastructure to solve the mentioned problems. For example, there is a hook where a message can be implemented in a particular class and such message will be send once the materialization is done. In such method one can implement the necessary behavior to get a meaningful object. For instance, a new socket may be asked and assigned. Nevertheless sometimes there is nothing to do, *e.g.,* if the file of the file stream was removed by the operating system.

It is worth to note here that some well known special objects are treated as external references, because that is the expected behavior for a serializer. Some examples are Smalltalk, Transcript and Processor.

Portability. As we explained in other sections, portability is not our main focus. Thus, the only portability aspect we are interested in is between Fuel versions. Below we talk about our versioning mechanism.

Versatility and Abstraction capacity. Both concerns are tightly tied. The goal is to be as concrete and exact as possible in the graph recreation, but providing flexible ways to customize abstractions as well as other kinds of substitutions in the graph. This last affirmation still has to be implemented.

Security. Our goal is to give the user the possibility to configure validation rules to be applied over the graph (ideally) before having any secondary effect on the environment. This is not yet implemented.

Atomicity. Unfortunately, Fuel can have problems if the graph changes during the analysis or serialization phase. This is an issue we have to work on in next versions.

7.2 Fuel serializer challenges

In this section, we explain how Fuel implements some of the features previously commented. There are some mentioned challenges we do not include here because Fuel does not support them at the moment.

Maintaining identity. Although it can be disabled, the default behavior when traversing the graph is to recognize some objects as *external references*: Classes registered in Smalltalk, global objects *i.e.,* referenced by global variables, global bindings *i.e.,* included in Smalltalk globals associations, and class variable bindings *i.e.,* included in the classPool of a registered class.

It is worth to mention that this mapping is done at object granularity, *e.g.,* not every instance of Class will be recognized as external. If a class is not in Smalltalk globals or if it has been specified as an *internal class*, it will be traversed and serialized in full detail.

Transient values. Fuel supports this by the class-side hook fuelIgnoredInstanceVariableNames. The user can specify there a list of variable names whose values will not be traversed or serialized. On materialization they will be restored as nil.

Cyclic object graphs and duplicates. Fuel checks that every object of the graph is visited once, supporting both cycle and duplicate detection.

Class shape change tolerance. Fuel stores the list of variable names of the classes that have instances in the graph being written. While recreating an object from the stream, if its class (or anyone in the hierarchy) has changed, then this meta information serves to automatically adapt the stored instances. When a variable does not exist anymore, its value is ignored. If a variable is new, it is restored as nil.

Versioning. This is provided by Fuel through a experimental implementation that wraps the standard Fuel format, appending at the beginning of the stream a version number. When reading, if such number matches with the current version, it is straightforward materialized in the standard way. If it does not match, then another action can be taken depending on the version. For example, suppose the difference between the saved version and the current version is a cluster that has been optimized and so their formats are incompatible. Now, suppose the wrapper has access to the old cluster class and so it configures the materializer to use this implementation instead of the optimized one. Then, it can adapt the way it reads, providing backward compatibility.

Graphical progress update. Fuel has an optional package that is used to show a progress bar while processing either the analysis, the serialization or the materialization. This *GUI* behavior is added via subclassification of Fuel core classes, therefore it does not add overhead to the standard *non-interactive* use case. This implementation is provisional since the future idea is to apply a *strategy design pattern*.

8. Benchmarks

We have developed several benchmarks to compare different serializers. To get meaningful results all benchmarks have to be run in the same environment. In our case we run them in the same Smalltalk dialect, with the same Virtual Machine and same Smalltalk image. Fuel is developed in Pharo, and hence the benchmarks were run in Pharo Smalltalk, image version Pharo-1.3-13257 and Cog Virtual Machine version "VMMaker.oscog-eem.56". The operating system was Mac OS 10.6.7.

In the following benchmarks we have analyzed the serializers: Fuel, SIXX, SmartRefStream, ImageSegment, Magma object database serializer, StOMP and SRP. Such serializers are explained in Section 9.

8.1 Benchmarks Constraints and Characteristics

Benchmarking software as complex as a serializer is difficult because there are multiple functions to measure which are used independently in various real-world use-cases. Moreover, measuring only the speed of a serializer is not complete and it may not even be fair if we do not mention the provided features of each serializer. For example, providing a hook for user-defined reinitialization action after materialization, or supporting class shape changes slows down serializers.

Here is a list of constraints and characteristics we used to get meaningful benchmarks:

All serializers in the same environment. We are not interested in compare speed with serializers that do not run in Pharo.

Use default configuration for all serializers. Some serializers provide customizations to improve performance, *i.e.,* some parameters or settings that the user can set for serializing a particular object graph. Those settings would make the serialization or materialization faster. For example, a serializer can provide a way to do *not* detect cycles. Detecting cycles takes time and memory hence, not detecting them is faster. Consequently, if there is a cycle in the object graph to serialize, there will be a loop and finally a system crash. Nevertheless, in certain scenarios the user may have a graph that he knows that there is no cycles.

Streams. Another important point while measuring serializers performance is which stream to be used. Usually, one can use memory based stream based and file based streams. Both measures are important and there can be significant differences between them.

Distinguish serialization from materialization. It makes sense to consider different benchmarks for the serialization and for the materialization.

Different kind of samples. Benchmark samples are split in two kinds: primitive and large. Samples of primitive objects are samples with lots of objects which are instances of the same class and that class is "primitive". Examples of those classes are Bitmap, Float, SmallInteger, LargePositiveInteger, LargeNegativeInteger, String, Symbol, WideString, Character, ByteArray, etc. Large objects are objects which are composed by other objects which are instances of different classes, generating a large object graph.

Primitive samples are useful to detect whether one serializer is better than the rest while serializing or materializing certain type of object. Large samples are more similar to the expected user provided graphs to serialize and they try to benchmark examples of real life object graphs.

Avoid JIT side effects. In Cog (the VM we used for benchmarks), the first time a method is used, it is executed in the standard way and added to the method cache. The second time the method is used, that means, when it is found in the cache, Cog converts that method to machine code. However, extra time is needed for such task. Only the third time the method will be executed as machine code and without extra effort.

It is not fair to run sometimes with methods that has been converted to machine code and sometimes with methods that have not. Therefore, for the samples we first run twice the same sample without taking into account its execution time to be sure to be always in the same condition. Then the sample is finally executed and its execution time is computed.

8.2 Benchmarks serializing with memory based streams

In this benchmarks we use memory based streams.

Primitive objects serialization. Figure 5 shows the results of primitive objects serialization.

Figure 6 shows the materialization of the serialization done for Figure 5.

The conclusions for serializing primitive objects with memory based streams are:

- We did not include SIXX in the charts because it was so slow that otherwise we were not able to show the differences between the rest of the serializers. This result is expected since SIXX is a text based serializer, which is far slower than a binary one. However, SIXX can be opened and modified by any text editor. This is an usual trade-off between text and binary formats.
- Magma and SmartRefStream serializers seem to be the slowest ones most of the cases.
- StOMP is the fastest one in serialization. Near to them there are Fuel, SRP, and ImageSegment.

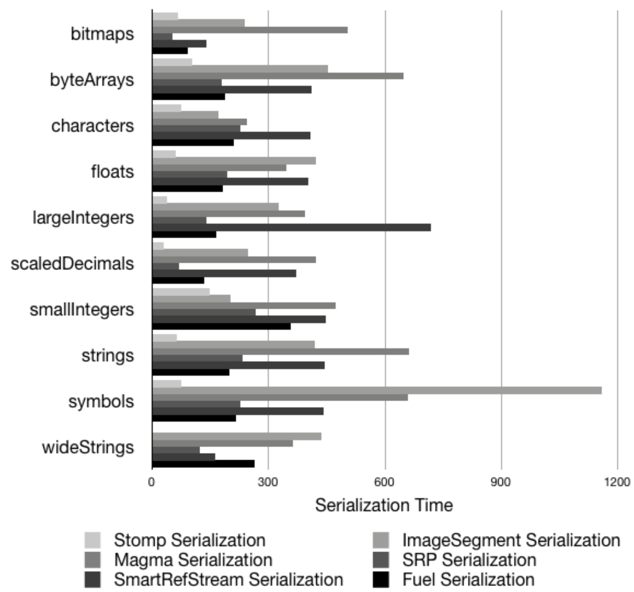

Figure 5. Primitive objects serialization in memory.

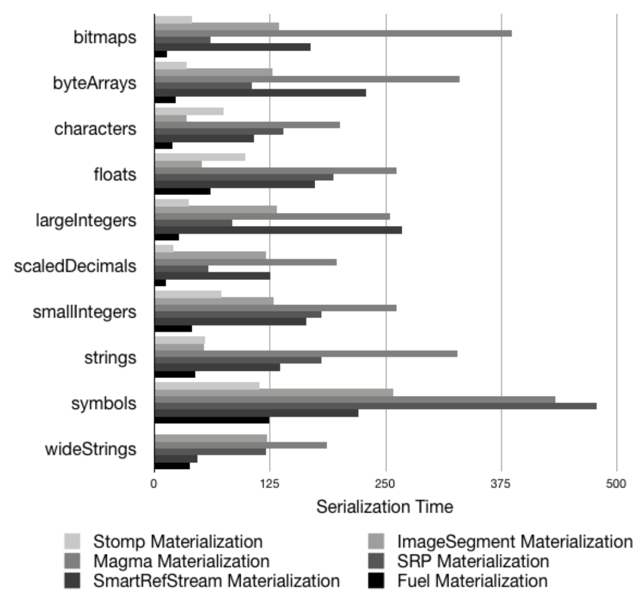

Figure 6. Primitive objects materialization from memory.

- Magma serializer is slow with “raw bytes” objects such as Bitmap and ByteArray, etc.
- Most of the cases, Fuel is faster than ImageSegment, which is even implemented in the Virtual Machine.
- ImageSegment is really slow with Symbol instances. We explain the reason later.
- StOMP has a zero (its color does not even appear) in the WideString sample. That means that cannot serialize those objects.
- In materialization, Fuel is the fastest one. Then after there are StOMP and ImageSegment.

Large objects serialization. As explained, these samples contain objects which are composed by other objects which are instances of different classes, generating a large object graph. Figure 7 shows the results of large objects serialization. Such serialization is also done with memory based stream. Figure 8 presents the materialization results when using the same scenario of Figure 7.

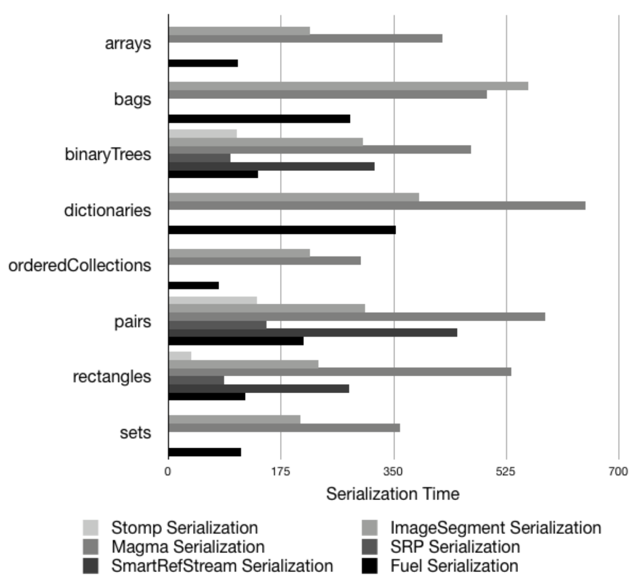

Figure 7. Large objects serialization in memory.

The conclusions for large objects are:

- The differences in speed are similar to the previous benchmarks. This means that whether we serialize graphs of all primitive objects or objects instances of all different classes, Fuel is the fastest one in materialization and one of the best ones in serialization.
- StOMP, SRP, and SmartRefStream cannot serialize the samples for *arrays*, *orderedCollections*, *sets*, etc. This is because those samples contain different kind of objects, included BlockClosure and MethodContext. This demonstrates that the mentioned serializers does not support serialization and materialization of all kind of objects. At least, not out-of-the-box.

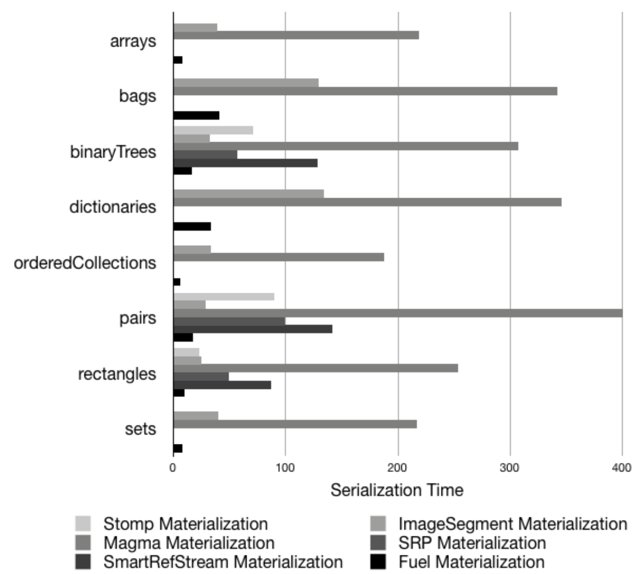

Figure 8. Large objects materialization from memory.

8.3 Benchmarks serializing with file based streams

Now we use file based streams. In fact, the exact stream we use is MultiByteFileStream. Figure 9 shows the results of primitive objects serialization.

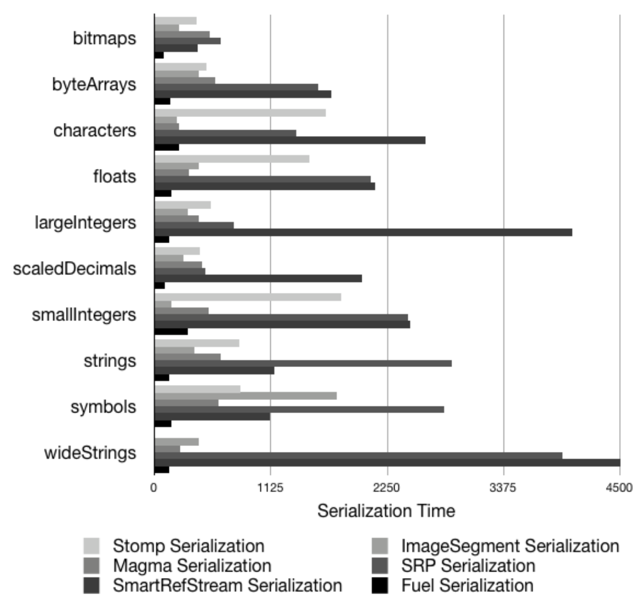

Figure 9. Primitive objects serialization in file.

Figure 10 shows the same scenario of Figure 9 but the results of the materialization.

The conclusions for serialization with file based streams are:

- It is surprising the differences between serializing in memory and in file. In serialization, SmartRefStream is by far the slowest.

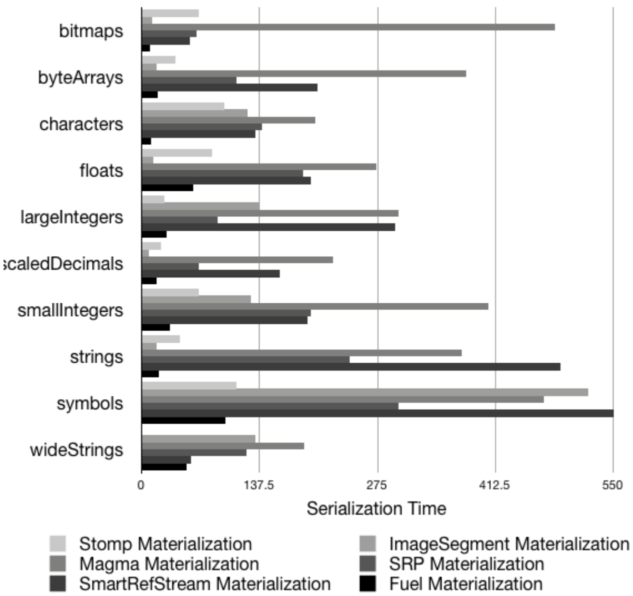

Figure 10. Primitive objects materialization from file.

- SRP and StOMP have good performance when serializing in memory, but not at all when serializing to a file based stream.
- Fuel is the fastest one, taking advantage of its internal buffering technique.
- Magma was one of the slowest in memory based stream but in this case it is much better.
- SmartRefStream and SRP are really slow with WideString instances.
- ImageSegment is slow with Symbol instances.

The conclusions for materialization with file based streams are:

- Again, Magma serializer is slow with "raw bytes" objects such as Bitmap and ByteArray, etc.
- ImageSegment is slow with Symbol instances.

These benchmarks showed that different serializers perform differently when serializing in memory or in files.

8.4 ImageSegment Results Explained

ImageSegment seems to be really fast in certain scenarios. However, it deserves some explanations of how ImageSegment works. Basically, ImageSegment receives the user defined graph and it needs to distinguish between *shared objects* and *inner objects*. Inner objects are those objects inside the subgraph which are *only* referenced from objects inside the subgraph. *Shared objects* are those which are not only referenced from objects inside the subgraph, but also from objects outside.

All *inner objects* are put into a byte array which is finally written into the stream using a primitive implemented in the virtual machine. After, ImageSegment uses SmartRefStream to serialize the *shared objects*. ImageSegment is fast mostly because it is implemented in the virtual machine. However, as we saw in our benchmarks, SmartRefStream is not really fast. The real problem is that it is difficult to control which objects in the system are pointing to objects inside the subgraph. Hence, most of the times there are several *shared objects* in the graph. The result is that the more *shared objects* there are, the slower ImageSegment is because those *shared objects* will be serialized by SmartRefStream.

All the benchmarks we did with primitive objects (all but Symbol) take care to create graphs with zero or few shared objects. That means that we are measuring the fastest possible case ever for ImageSegment. Nevertheless, in the sample of Symbol one can see in Figure 5 that ImageSegment is really slow in serialization, and the same happens with materialization. The reason is that in Smalltalk all instances of Symbol are unique and referenced by a global table. Hence, all Symbol instances are shared and therefore, serialized with SmartRefStream.

We did an experiment where we build an object graph and we increase the percentage of *shared objects*.

Figure 11 shows the results of primitive objects serialization with file based stream. Axis *X* represents the percentage of shared objects inside the graph and the axis *Y* represents the time of the serialization.

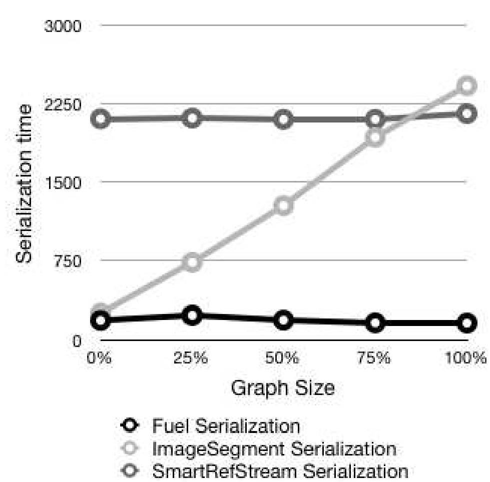

Figure 11. ImageSegment serialization in presence of shared objects.

Figure 12 shows the same scenario of Figure 11 but the results of the materialization.

Conclusions for ImageSegment results

- The more shared objects there are, the more ImageSegment speed is similar to SmartRefStream.
- For materialization, when all are shared objects, ImageSegment and SmartRefStream have almost the same speed.
- For serialization, when all are shared objects, ImageSegment is even slower than SmartRefStream. This is be-

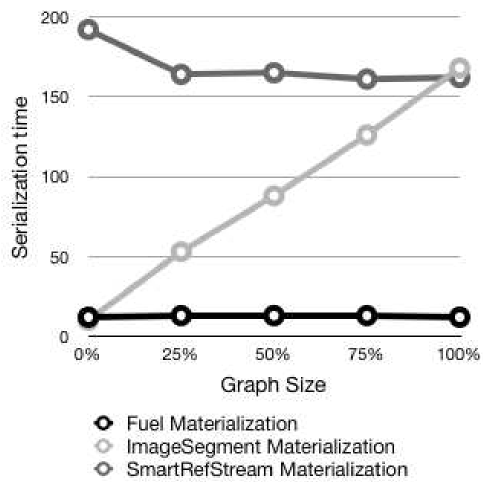

Figure 12. ImageSegment materialization in presence of shared objects.

cause ImageSegment needs to do the whole memory traverse anyway to discover shared objects.

- ImageSegment is unique in the sense that its performance depends in both: 1) the amount of references from outside the subgraph to objects inside; 2) the total amount of objects in the system, since the time to traverse the whole memory depends on that.

8.5 General Benchmarks Conclusions

Magma serializer seems slow when serializing in memory, but it is acceptable taking into account that this serializer is designed for a particular database. Hence, the Magma serializer does extra effort and stores extra information that is needed in a database scenario but may not be necessary for any other usage.

SmartRefSteam provides a good set of hook methods for customizing serialization and materialization. However, it is slow and its code and design are not good from our point of view. ImageSegment is known to be really fast because it is implemented inside the virtual machine. Such fact, together with the problem of *shared objects*, brings a large number of limitations and drawbacks as it has been already explained. Furthermore, with Cog we demonstrate that Fuel is even faster in both, materialization and serialization. Hence, the limitations of ImageSegment are not worth it.

SRP and StOMP are both aimed for portability across Smalltalk dialects. Their performance is good, mostly at writing time, but they are not as fast as they could because of the need of being portable across platforms. In addition, for the same reason, they do not support serialization for all kind of objects.

This paper demonstrates that Fuel is the fastest in materialization and one the fastest ones in serialization. In fact, when serializing to files, which is what usually happens, Fuel is the fastest. Fuel can also serialize any kind of object. Fuel aim is not portability but performance. Hence, all the results make sense from the goals point of view.

9. Related work

The most common example of a serializer is one based on XML like SIXX [17] or JSON [9]. In this case the object graph is exported into a portable text file. The main problem with text-based serialization is encountered with big graphs as it does not have a good performance and it generates huge files. Other alternatives are ReferenceStream or SmartReferenceStream. ReferenceStream is a way of serializing a tree of objects into a binary file. A ReferenceStream can store one or more objects in a persistent form including sharing and cycles. The main problem of ReferenceStream is that it is slow for large graphs.

A much more elaborated approach is Parcel [12] developed in VisualWorks Smalltalk. Fuel is based on Parcel's pickling ideas. Parcel is an atomic deployment mechanism for objects and source code that supports shape changing of classes, method addition, method replacement and partial loading. The key to making this deployment mechanism feasible and fast is a pickling algorithm. Although Parcel supports code and objects, it is more intended to source code than normal objects. It defines a custom format and generates binary files. Parcel has good performance and the assumption is that the user may not have a problem if saving code takes more time, as long as loading is really fast.

The recent StOMP[1] (Smalltalk Objects on MessagePack[2]) and the mature SRP[3] (State Replication Protocol) are binary serializers with similar goals: Smalltalk-dialect portability and space efficiency. They are quite fast and configurable, but they are limited with dialect-dependent objects like BlockClosure and MethodContext. Despite the fact that their main goals differ from ours, we should take into account their designs.

Object serializers are needed and used not only by final users, but also for specific type of applications or tools. What it is interesting is that they can be used outside the scope of their project. Some examples are the object serializers of Monticello2 (a source code version system), Magma object database, Hessian binary web service protocol [7] or Oracle Coherence*Web HTTP session management [13].

Martinez-Peck et al. [11] performed an analysis of ImageSegment (a virtual machine serialization algorithm) and they found that the speed increase in ImageSegment is mainly because it is written in C compared to other frameworks written in Smalltalk. However, ImageSegment is slower when objects in the subgraph to be serialized are externally referenced.

[1] http://www.squeaksource.com/STOMP.html

[2] http://msgpack.org

[3] http://sourceforge.net/projects/srp/

10. Conclusion and Future Work

In this paper, we have looked into the problem of serializing object graphs in object oriented systems. We have analyzed its problems and challenges. What is important is that these steps, together with their problems and challenges, are general and they are independent of the technology.

These object graphs operations are important to support virtual memory, backups, migrations, exportations, etc. Speed is the biggest constraint in these kind of graph operations. Any possible solution has to be fast enough to be actually useful. In addition, this problem of performance is the most common problem among the different solutions. Most of them do not deal properly with it.

We presented Fuel, a general purpose object graph serializer based on a pickling format and algorithm different from typical serializers. The advantage is that the unpickling process is optimized. In one hand, the objects of a particular class are instantiated in bulk since they were carefully sorted when pickling. In the other hand, this is done in an iterative instead of a recursive way, what is common in serializers. The disadvantage is that the pickling process takes an extra time in comparison with other approaches. Besides, we show in detailed benchmarks that we have the best performance in most of the scenarios we are interested in.

We implement and validate this approach in the Pharo Smalltalk environment. We demonstrate that it is possible to build a fast serializer without specific VM support, with a clean object-oriented design, and providing most possible required features for a serializer.

Even if Fuel has an excellent performance and provided hooks, it can still be improved. Regarding the hooks, we would like to provide one that can let the user replace one object by another one, which means that the serialized graph is not exactly the same as the one provided by the user.

Instead of throwing an error, it is our plan to analyze the possibility of create light-weight shadow classes when materializing instances of an inexistent class. Another important issue we would like to work on is in making everything optional, *e.g.,* cycle detection. Partial loading as well as being able to query a serialized graph are concepts we want to work in the future.

To conclude, Fuel is a fast object serializer built with a clean design, easy to extend and customize. New features will be added in the future and several tools will be build on top of it.

Acknowledgements

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER through the CPER 2007-2013.

References

[1] J. K. Bennett. The design and implementation of distributed Smalltalk. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 318–330, Dec. 1987.

[2] A. P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[3] M. D. Bond and K. S. McKinley. Tolerating memory leaks. In G. E. Harris, editor, *OOPSLA: Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 109–126. ACM, 2008.

[4] F. Breg and C. D. Polychronopoulos. Java virtual machine support for object serialization. In *Joint ACM Java Grande - ISCOPE 2001 Conference*, 2001.

[5] P. Butterworth, A. Otis, and J. Stein. The GemStone object database management system. *Commun. ACM*, 34(10):64–77, 1991.

[6] D. Decouchant. Design of a distributed object manager for the Smalltalk-80 system. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 444–452, Nov. 1986.

[7] Hessian. http://hessian.caucho.com.

[8] Java serializer api. http://java.sun.com/developer/technicalArticles/Programming/serialization/.

[9] Json (javascript object notation). http://www.json.org.

[10] T. Kaehler. Virtual memory on a narrow machine for an object-oriented language. *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, 21(11):87–106, Nov. 1986.

[11] M. Martinez Peck, N. Bouraqadi, M. Denker, S. Ducasse, and L. Fabresse. Experiments with a fast object swapper. In *Smalltalks 2010*, 2010.

[12] E. Miranda, D. Leibs, and R. Wuyts. Parcels: a fast and feature-rich binary deployment technology. *Journal of Computer Languages, Systems and Structures*, 31(3-4):165–182, May 2005.

[13] Oracle coherence. http://coherence.oracle.com.

[14] Pickle. http://docs.python.org/library/pickle.html.

[15] Google protocol buffers. http://code.google.com/apis/protocolbuffers/docs/overview.html.

[16] R. Riggs, J. Waldo, A. Wollrath, and K. Bharat. Pickling state in the Java system. *Computing Systems*, 9(4):291–312, 1996.

[17] Sixx (smalltalk instance exchange in xml). http://www.mars.dti.ne.jp/~umejava/smalltalk/sixx/index.html.

[18] D. Ungar. Annotating objects for transport to other worlds. In *Proceedings OOPSLA '95*, pages 73–87, 1995.

[19] D. Wiebe. A distributed repository for immutable persistent objects. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 453–465, Nov. 1986.